SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 57–76 (1980)

Development of the ZED Text Editor

PHILIP HAZEL

Computer Laboratory, Corn Exchange Street, University of Cambridge, England

SUMMARY

The development process which resulted in a new text editor is described. The reasons for undertaking this development are discussed, and in particular the increased amount of non-interactive use of text editors is stressed. The approach taken to the project is described, including the involvement of users in defining the specification and testing prototypes. Finally the main features of the new editor, ZED, are covered.

KEY WORDS Text editor Text processing OS/360 EDIT

INTRODUCTION

The IBM 370/165 at Cambridge provides batch and timesharing services for academic users. Early in the life of this machine a general-purpose text editor was developed which would run in both the timesharing and batch environments. This program is called EDIT.[1] Its facilities are based on a previous text editor developed in Cambridge for the Titan computer.[2]

From the time that EDIT was put into service, users continued to suggest extensions and enhancements. Some of these were included in the program, but after a time modification became more and more difficult as the original design was stretched to its limit. Certain inadequacies were impossible to correct, and finally a major redesign was undertaken. The result was a completely new text editor, called ZED.

The following sections describe some of the inadequacies of EDIT, and why it was not possible to overcome them without a major redesign, the considerations involved in the design of ZED, the implementation and testing process, and ZED itself.

INADEQUACIES OF EDIT

EDIT was conceived as an interactive editor, with non-interactive use a lesser activity, mainly required for simple tasks such as incremental editing. However, more and more users began making use of EDIT as a general text manipulation language by means of stored sequences of commands which were run on a regular basis. One example of such use is the maintenance of a list of members of a society, together with associated information such as type of membership, address, telephone number, etc. Provided the various fields are delimited by appropriate character strings, EDIT can be used to extract subsets of this data, such as all the members in a given town. Another common use is the post-processing of output from other programs.

Using a text editor in this way may not always be the most efficient solution to the problem in terms of machine resources; however, it is very efficient in human resources because the user is working within a familar system where all the data is immediately

0038–0644/80/0110–0057$01.00

Received 20 July 1979
Revised 21 September 1979

readable and debugging can be carried out interactively. These are the same benefits that can be obtained by using an interactive programming language such as APL for computational problems. A text editor can also be used to implement quick experimental versions of text processing algorithms before investing resources in full scale implementations.

While it is possible to do the sort of processing described above using EDIT, in many cases the command sequences are very clumsy. There are three main areas in which EDIT facilities are lacking: searching, command control and line handling.

Searching

In EDIT, the only facilities for searching for a particular line of text are the F and L commands, which search for a line beginning with or containing a given string, respectively. The ability to find a line NOT containing a given string, or containing any one of a number of strings, and the ability to find empty lines were the most often requested extensions.

Command control

EDIT contains no explicit control commands. Users wanting to test for the presence of a string in a line have to do a dummy alteration to the line, and ignore the resulting error when the string is not present. This is unsafe as well as clumsy. Explicit control commands such as IF and WHILE were frequently suggested.

The restriction that a repeated group of EDIT commands must be all on one line (originally imposed for efficiency reasons) diminished the usefulness of this feature.

Line handling

EDIT deals with one line of source text at a time and always moves forwards through a source file. The restriction to forward movement was imposed for efficiency and to minimize the amount of store used. In practice many operations require two passes through the file because of this restriction, thus nullifying any efficiency gains. Consequently an enhancement that allowed cheap backward movement was desirable.

A related facility is the movement of a line or lines from one part of a file to another. In EDIT this must be done by writing out the lines to be moved to a disc file and then reading them back again which is a comparatively expensive operation.

UPGRADING CONSIDERATIONS

This section summarizes the reasoning behind the decision to implement a new editor in the same style as EDIT, and describes how the design was carried out.

Full screen editors

Since EDIT was first implemented, terminals with screen displays have become commonplace on all types of processor. It is widely agreed that the most humanly acceptable form of interactive text editing is by means of a full screen editor which displays a page at a time and allows the user to point, by means of a light pen or cursor, at the portions of text to be amended. Such an editor requires a device capable of operating at high speed in page mode.

The provision of a full screeen text editor as an upgrade to EDIT was not considered, for the following reasons:

1. It would be purely interactive rather than general purpose; the deficiencies of EDIT are felt most in non-interactive applications.
2. There are no full screen devices currently attached to the 370/165, and in any case such devices are likely to remain in a minority because of cost.

Imported editors

Having decided that a more up-to-date editor was desirable, one approach would have been to have acquired an existing editor from elsewhere. The reasons for not looking for such an editor were twofold. Firstly, it was very important to preserve as much compatibility with EDIT as possible. There are currently over 2000 registered users of the 370/165, many of whom have been using editors in the style of EDIT for more than a decade.[5]

Secondly, the unique problems of running under the OS/360 operating system made it highly unlikely that software from elsewhere would adhere to local utility standards. An editor was required which would run sensibly both interactively and in the batch, and would be capable of reading and writing any sequential file in the system.

The last point requires some elaboration for the benefit of readers not familiar with OS/360. Under this operating system sequential files may be in one of three formats: fixed length records, variable length records or undefined length records. In addition, the first two types may be blocked or unblocked, and variable length records may, optionally, span physical blocks. The record lengths and block sizes may be chosen arbitrarily by the user, within the physical limits of the peripheral device. A further complication is that the first character of the records in any given file may optionally be designated as a control character, and there are two possible kinds of control character.

At the level of the interface to the operating system, the programmer must be aware of all the possible formats for sequential files, and must write code to deal with each of them. Special code is also required for driving terminals in an interactive fashion. A consequence of this is that many Assembler programs and high level language systems support only a subset of the available formats. A truly general purpose editor, however, must be capable of dealing with all of them, and must also be capable of converting input in one format to output in another.

A further implementation restriction was imposed by the fact that the 370/165 does not have paging hardware, and hence does not offer virtual memory facilities. The editor therefore has to make minimal demands on main storage, and is required to be re-entrant in order to reduce the swapping overhead for interactive users.

The design process

The first planning document proposing a major upgrade to EDIT attempted as far as possible to keep a compatible syntax for editing commands. During the discussion which followed it became clear that it was impossible to provide all the required enhancements within the old syntax. One of the main problems was that EDIT allows any character as a delimiter for text strings. Thus the F command, which finds a line beginning with a string, could for example be written

```
FFABCF
```

in order to find the string ABC. As a result of this delimiter rule only one command can begin with the letter F. This can, of course, easily (and only slightly incompatibly) be overcome by forbidding the use of letters as delimiters. There still remain problems

with special characters if, for example, a way of specifying 'the last search string' is required. The use of the space character as a delimiter causes difficulties when strings are given attributes represented by letters, and these must be separated from the command name by spaces.

The result of the planning discussion was a decision to design a completely new text editor, retaining compatibility only where convenient. The processes of design and implementation were then carried out in parallel, experience of using the first primitive, incomplete versions being fed back into the design.

The development of the user specification was coordinated by one person. However, the many drafts that were produced were widely circulated and comments were received from many sources. The implementation was entirely a one-person affair. The elapsed time from start to first general release was one year; the estimated time spent on the program was six months. Very early in the implementation cycle primitive versions of ZED were made available to selected users for evaluation. The first of these consisted of little more than the F command. The feedback from this experimental use was invaluable in detecting poor design choices as well as program bugs. A number of changes were made to the syntax of editing commands, some of them fairly substantial, and several commands were respecified as a result of user comments. Such modifications were possible because ZED was developed in an environment which encourages flexibility. In addition, as it was a redevelopement of an existing facility, there was no deadline for its completion. The result of this parallel design and implementation is a program whose rougher edges had been removed before release to the general user population.

MAIN FEATURES OF ZED

This section describes the most important facilities provided by ZED in an informal manner. A complete list of editing commands (including many not described in this paper) is given as an appendix, while a full specification of the program appears in Reference 3.

ZED commands consist of a command name, which is either a sequence of letters or a single special character, optionally followed by arguments. Seven different types of argument exist: strings, qualified strings, search expressions, names, numbers, switch values and command groups. Commands are delimited by end of line (except within parentheses) or the semicolon character. Spaces may occur between a command name and the first argument, between non-string arguments, and between commands. A space is only necessary in these places to separate two successive items which would otherwise be treated as one (e.g. two numbers). The character \ is used to introduce comment, terminated by end of line.

Many commands may be obeyed a fixed number of times by preceding them by a count. For example,

```
3N
```

obeys the N command (which moves to the next line) three times. A sequence of commands may be grouped together by enclosing it in parentheses. Such command groups may optionally be preceded by a count, may extend over more than one line of command input, and may be nested to any depth. Command groups are used as arguments for the control commands.

String matching

A string in ZED consists of an arbitrary sequence of characters enclosed in matching delimiters which do not themselves appear in the string. The delimiter character for any given string must be one of

```
+ - * / . , ? ! : ' "
```

Some examples of strings are

```
/The cat/   :A = A+B(I):   " -/"
```

In the examples which follow, the solidus character / is used as the string delimiter.

The basic unit of string matching in ZED is the qualified string, which is a string preceded by a sequence of qualifying letters. These place constraints on the matching process. A string with no qualifiers will match anywhere in a line of source text, but a string preceded by B will match only at the beginning of a line. Thus for example,

```
F /Jabberwocky/
```

will find a line containing 'Jabberwocky', but

```
F B/slithy toves/
```

will only find a line which *begins* with 'slithy toves'. Similarly, a string preceded by E will only match at the *end* of a line, while a string preceded by P will only match a line containing *precisely* the given characters and nothing else. If P is used with a null string it matches an empty line.

The qualifier N can be used in conjunction with other qualifiers, and it negates the result of the matching test. Thus

```
F NB/The sun was shining/
```

finds a line that does *not* begin with 'The sun was shining'. Note that this is not the same as a line containing 'The sun was shining' not at the beginning.

```
F NP//
```

finds a non-empty line. The qualifier W specifies a *word* search, which means that the string that is matched must neither be preceded nor followed by a letter or a digit, while the qualifier U specifies that the matching is to take place as though all the letters involved were *uppercased*. Thus

```
F UW/king/
```

will find lines containing the words 'king', 'KING' or even 'KiNg', but not 'kingpin' or 'thinking'.

The qualifier S specifies that lines are to be considered as starting at the first significant (non-space) character. This is most often used in conjunction with the B or P qualifiers when editing indented text. For example,

```
F SB/X: = /
```

If a string occurs more than once in a line, the second or subsequent occurrences can be matched by specifying a numerical qualifier. For example,

```
2/cat/
```

matches the second occurrence of 'cat' in a line, and hence

F 2/cat/

finds a line containing at least two occurrences of the string 'cat'.

A matching operation can be restricted to certain character postions in a line by means of the column qualifier.

F [73,80]/ABCD/

searches for a line containing 'ABCD' in columns 73–80.

As well as their use in the F command, as illustrated in the above examples, qualified strings are also used to indicate a position in the current line at which textual changes are to be made. In these cases the N qualifier is not allowed. For example,

B UW/king/Red/

causes the text 'Red' to be inserted *before* that point in the current line matched by UW/king/. The beginning and the end of the line can be indicated by the qualified strings

// and E//

respectively. There is also a qualifier L which causes the search for the string to occur leftwards, thus ensuring that the *last* occurrence in the line is matched. If the current line were

If seven maids with seven mops

then the command

B L/seven/twenty-/

would turn it into

If seven maids with twenty-seven mops

A numerical qualifier can be used with L to match the *n*th last occurrence of a string.

Other commands for operating on the current line are A (insert string after), E (exchange string), DFA, DFB (delete from after, before), DTA, DTB (delete till after, before), LC, UC (force lower, upper case) and SA, SB (split line after, before).

Whenever a current line is first altered by means of one of the above commands, a copy of the original line is taken. At any time before the line ceases to be current the original line can be restored by means of the UNDO command. This provides a convenient means of recovery from typing errors when editing interactively.

Search expressions

Qualified strings may be combined using the operators & (logical and) and | (logical or) to form search expressions. Search expressions are always enclosed in parentheses, may contain nested parentheses, and may extend over more than one line of command input. They are used as arguments to commands which search for or test lines of the source. Commands which perform some action on the current line are restricted to a single qualified string argument, as described in the previous section.

The command

F (/SUBROUTINE/ | /FUNCTION/)

finds a line containing 'SUBROUTINE' or 'FUNCTION' (or both), whereas

```
F (B/king/ & (NE/queen/ | /knave/))
```

finds a line beginning with 'king' and not ending with 'queen' or beginning with 'king' and containing 'knave'.

The only restriction on the size and complexity of search expressions is the amount of main store available to ZED.

The last search expression

The last obeyed search expression is re-accessed in a subsequent command if & is written for the argument. For example,

```
F /Tweedledum/; N; F &
```

finds a line containing 'Tweedledum', moves to the next line, then obeys the second F command using the same argument as the first. Commands such as F which take only a single search expression argument can in fact be written without arguments, in which case & is assumed.

The N command used in the above example is necessary because F always starts its search at the current line. It was recognized during the design of ZED that this is not an ideal specification for an interactive editor, where the user is unlikely to type redundant F commands. However, in non-interactive applications, particularly in cases where files of editing commands are stored and updated, the required line may already be current at the start of an F command as the result of some previous, unrelated editing operation, and should therefore be included in the search.

In addition to its use with the F command, & can be used for commands which operate on the current line and which require a qualified string argument. For example,

```
F /Tweedledum/; E&/Tweedledee/
```

finds a line containing 'Tweedledum' and then obeys

```
E/Tweedledum/Tweedledee/
```

to exchange the two strings. This use of & is only valid when the most recently obeyed search expression succeeded, and in succeeding defined a unique position in the line. Thus

```
F (/red/ | /white/); E&/purple/
```

is a valid use of &, because whichever string is found will define the point of exchange. However,

```
F (/red/ & /white/); E&/purple/
```

is invalid, as is

```
F N/red/; E&/purple/
```

Control commands and loops

ZED contains a flexible set of control and loop commands, which include IF, UL (unless), WH (while), UT (until), IFEOF (if end of file), ULEOF (unless end of file)

and UTEOF (until end of file). The first four apply a search expression to the current line to test which control path to follow. For example,

```
IF (/cat/ | /dog/) THEN (
  B[73, 80]//animal/; ?) ELSE D
```

tests the current line for one of the strings 'cat' or 'dog'. If either is found the string 'animal' is inserted in columns 73–80 and the line is verified (? command); if neither is found the ELSE branch is taken and the current line is deleted.

A series of tests on the same line may be written using the connectors ELIF (else if) and/or ELUL (else unless). This reduces the number of parentheses required and makes the commands easier to read.

The loop commands repeatedly obey their command group argument while or until the search expression matches the current line. The scan for the elements of the search expression starts at the beginning of the line for each iteration. There are, however, windowing commands for setting a logical line beginning, and these may be used within the iterative command group to skip over contexts already dealt with. A simple example of the use of a loop command is

```
WH/  / E&/ /
```

which has the effect of changing all multiple spaces in the current line to a single space.

UTEOF (until end of file) is special in that it exits from the loop when an end of file error message would otherwise occur. Thus it can be used in sequences such as

```
UTEOF (F/unicorn/; B//***/; N)
```

which will eventually complete when the F command fails at end of file.

The normal flow of control in loops can be broken by means of the AGP (abandon group) command. This has the effect of exiting from the nearest textually surrounding level of command group parentheses, and any command of which the group is an argument. For example,

```
WH B/*/ (A/*/ /; N; IFEOF AGP)
```

inserts a space after the initial * of successive lines until either a line not beginning with * or the end of the source is reached. A repeat count can be used with AGP to exit from more than one level.

Movement through the source

The N command, which moves to the next line of the source text, has appeared in several previous examples. There is also a command P which moves back to the previous line. Similarly, corresponding to the F (find) command, which moves forwards, there is a BF (backward find) command which performs the same operation moving backwards through the source.

ZED implements backwards motion by keeping previous line in main store as long as possible before writing them out to the destination file. Backward motion to the start of the text is therefore not always possible for large sources. This restriction is imposed for reasons of efficiency. It means that no intermediate disc copies of the text are required in the majority of editing applications. The maximum number of lines held in store at one time depends only on the store available; on the Cambridge system where about 30K bytes of store is available for interactive use of ZED, around 500 lines can normally be held.

A number of other commands are available for traversing the source, either as their sole function or as a side-effect. Some of these take line numbers as arguments.

Each line that is read from the source is given a unique line number. Such lines are initially in ascending sequence of line number and are termed *original* lines. New inserted lines do not have line numbers, and lines which are moved around in the sequence (see the section *Line shuffling*) become *non-original*, and can no longer be referenced by line number. Hence commands with line number arguments can always determine from the argument in which direction to move.

Line shuffling

ZED provides 16 in-store buffers for the purpose of moving blocks of text around, and for replication. The TO command is used to specify the destination for lines which are passed in a forward direction. The default destination is a queue which is emptied onto the output file when its store is required, and at the end of editing.

```
TO BUFF n
```

directs lines to buffer n instead of the default queue;

```
TO
```

reverts to the default. The contents of any buffer other than that to which lines are currently directed may be inserted into the text before line m by the command

```
Im BUFF n
```

If m is omitted, the insertion occurs before the current line. A copy of the contents of a buffer can be inserted by the command

```
Im COPY n
```

This feature enables easy replication of blocks of text. The following example moves lines 10–15 to immediately before line 36:

```
M10          \move to line 10
TO BUFF 3    \select buffer 3
M16          \output lines 10–15
TO           \revert to normal output
I36 BUFF 3   \insert buffer 3 before line 36
```

This technique can also be used to move text to an earlier part of the source, provided that the insertion point is still in store and can be accessed by moving backwards.

The system of input and output queues and in-store buffers may be visualized as a railway shunting yard where the TO command operates the points, and each wagon represents a line of text. This is depicted in Figure 1.

At the end of each siding representing an in-store buffer there is a tunnel which connects through to the insert siding and which is brought into use by the I command with a buffer argument. When COPY is used with I this tunnel generates a copy of the lines in the buffer instead of moving the lines themselves.

The lines in a buffer can also be processed one by one as current lines by selecting the buffer as a source using the FROM command.

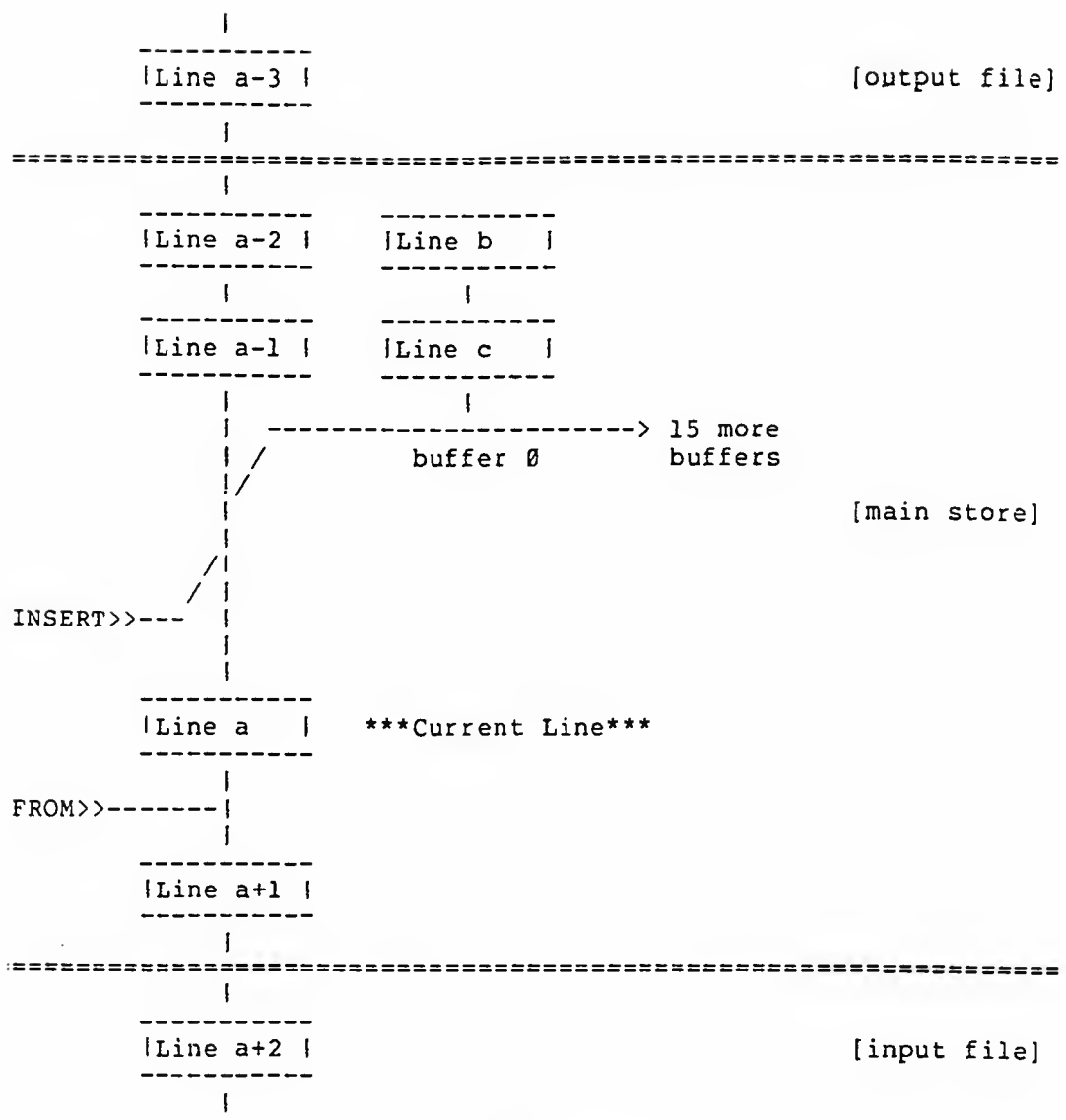

Figure 1. ZED railway analogy

FROM BUFF n

has the effect (in railway terms) of connecting a tunnel from the end of the siding for buffer n through to a point just before the current line. Commands which move forwards through the source then access the lines in the buffer. A dummy end of file is generated when the buffer is empty, and the command

FROM

may be used to revert to the normal source file.

Input and output selection

As well as controlling the in-store buffers, the FROM and TO commands can be used to select different input and output files. In this case the argument is a string which is an operating system dependent file name. For example,

FROM /DD1/

In the case of the TO command the queue of output lines is written to the output file before the switch is made, and for both commands the old file is left open so that reselection can take place later. These facilities can be used to merge a number of different sources, to split up a file into a number of separate files, or to effect line shuffling when the quantity of text to be moved is too great to fit into an in-store buffer. For example,

```
M1000          \move to line 1000
TO /DD1/       \output to temporary file
M1501          \lines 1000–1500 queued for DD1
TO             \reselect normal output
CF /DD1/       \close file DD1
I3600 /DD1/    \insert before line 3600
```

Space compression

A frequent use of a text editor is for the removal of redundant spaces from text, for example when inspecting data intended for a line printer at a slow and/or narrow terminal. This can be done in ZED by means of the ON command (see the section *Global operations*) but because it is so common a special efficient facility is provided. The command

CS+

turns on space compression, which ensures that all leading and trailing spaces are removed, and all multiple spaces converted to a single space as source lines are read. The command

CS–

turns compression off.

Justification

EDIT contains facilities for very simple justification by ensuring that all lines contain the maximum possible number of words. Similar facilities are provided in ZED, but instead of action taking place immediately the relevant command is obeyed, the justification parameters are remembered and used whenever a current line is passed in a forward direction or a line is inserted. Thus justification takes place in parallel with other editing activities.

The justification commands are as follows:

JLn

Set justification line length to n. Justified lines will not be longer than n characters.

J+ J–

Enable/disable justification. The default state is with justification turned off. As in EDIT, indented lines, blank lines and overprinting lines are taken as the start of new

paragraphs. A sequence of indented lines can, however, be justified by using the command

```
RJn m
```

which restricts justification to columns n to m. Any lines which contain text outside these columns are not justified.

Global operations

ZED provides the ability to specify global operations on the source text which are carried out in parallel with other editing. The single global command of EDIT has been replaced by the three commands GA, GB and GE which act globally in the same way that the A, B and E commands act on a single line. Thus, for example,

```
GA/cat/fish/
```

changes every occurrence of 'cat' into 'catfish' in the current line and in all succeeding lines until the end of the source is reached or until the global command is cancelled or disabled. The command

```
GE/cat/catfish/
```

is equivalent. All the features of qualified strings are available in these commands:

```
GA UW[60,]/the/y/
```

changes all occurrences of the word 'the', in upper or lower case, following column 60, by adding the letter 'y'.

The facility provided by the G commands is adequate for global changes involving simple additions or exchanges. More complicated operations can be specified by means of the ON command, which takes as its arguments a search expression and a command group. Whenever a line that matches the search expression becomes current during forward motion, the command group is obeyed. Commands in the group are restricted to those that do not move to a different current line. For example

```
ON (BS/SUBROUTINE/ | BS/FUNCTION/) (I
C
C
Z
COMMENT /NEW ROUTINE/
?
)
```

causes two empty FORTRAN comment lines to be inserted before any statement beginning with SUBROUTINE or FUNCTION, outputs a comment to the verification file, and verifies the line. The command

```
ON B/*/ (WH/**/ E&/*/)
```

causes all multiple asterisks to be changed to a single asterisk in lines which begin with an asterisk. An ON command may also optionally specify an ELSE command group, which is obeyed whenever the search expression does *not* match a new current line.

Each global command, whether G type or ON type, is allocated a number when it is obeyed. This number can be used as an argument to the DG, EG and CG commands

which respectively disable, enable and cancel the relevant global operation. Omission of an argument causes all existing globals to be affected. For example, to apply a global exchange to lines 13–19 and all lines after the one containing precisely ⟨⟨⟩⟩ the following commands could be used:

```
M13                 \move to line 13
GE/cat/dog/         \set up global exchange
M19                 \move to line 19
DG1                 \disable global 1
ON P/⟨⟨⟩⟩/ EG1      \enable when ⟨⟨⟩⟩ encountered
```

The currently existing globals can be inspected by means of the SHG (show globals) command. As well as listing the global numbers and corresponding commands, the number of times that each global has matched is given.

Procedures and command files

ZED procedures provides a means of naming sequences of commands for later use. There are no facilities for providing arguments to procedures. The command

```
PROC SQ (WH/**/ E/**/*/)
```

sets up a procedure called SQ, which, when obeyed by

```
DO SQ
```

squashes the current line by removing multiple asterisks. Procedures may be called (though not defined) within other procedures, and may be called recursively. The SHPROC and CPROC commands are used to show and cancel procedures respectively.

A sequence of ZED commands may be read from a file other than the current command file by the command

```
C /filename/
```

The argument is an operating system dependent file name. When the file is exhausted, control reverts to the previous command file. The C command can also be used to read command lines from an in-store buffer by using one of the forms

```
C BUFF n
C COPY n
```

The former leaves the buffer empty while the latter leaves it intact. This facility means that a sequence of ZED commands is capable of building up further sequences of commands whose content is dependent on the source text being processed.

Non-Printing Characters

A source text that is being edited may contain characters that are not available on the terminal or other device that is being used for verification output. ZED provides a special verification command intended to be used on lines containing a small number of such characters. Whereas the normal verification command ? outputs the current line as it stands, the special verification command ! generates two lines of output. The first line is the current line with special characters replaced in one of two ways:

1. Special characters which can be input using the standard escape mechanism on the Cambridge communications system are replaced by the escape character @.

2. Other special characters are replaced by the first hexadecimal digit of their EBCDIC value.

The second line of output contains hyphens under upper case letters, the rest of the escape combination under any escape characters, the second hexadecimal digits of any other special characters, and spaces in all other positions. Typical verification output from the ! command might be

```
The King bought @0@14·50
  -   -          @5 $
```

The escape combinations @@ and @$ represent an @ and a £ character respectively, while the hexadecimal 05 is a tab character.

For editing binary data or data with a high proportion of non-printing characters, ZED can operate in hexadecimal mode. This is switched on and off by the X+ and X− commands. In this mode lines are verified in hexadecimal as a series of eight character groups, and command strings referring to textual data must also be expressed in hexadecimal. The line used in the above example would be verified as

E3888540 D2899587 408296A4 8788A340 7C054AF1 F44BF5F0

and to change 'The' to 'Their' the command

A/85/8889/

could be used.

Commands copied from EDIT

A number of EDIT's commands are reproduced without any essential change in ZED. These include the line number oriented insert, replace and delete commands (I, R and D), the commands for operating on individual characters (⟩, #, %, S) and the margin, window and fixed field commands (MA, MAV, RV, RF and K).[1]

IMPLEMENTATION

ZED is written in IBM 370 Assembler, for reasons of size and efficiency. It is, however, designed in a structured manner as a series of well-defined procedures with a standard interface convention. This discipline has proved very effective in simplifying the debugging process. The code of ZED occupies 36K bytes of main storage. A minimum of 8K bytes of working storage is required; if more is available ZED uses it for holding more of the source in main store.

The algorithms used in implementing ZED are mostly straightforward in nature, with one exception. The algorithm used for searching for a given string in a line of text is a modified version of that published by Boyer and Moore.[4] Their algorithm involves the use of a table of 256 bytes (for EBCDIC code); in the context of ZED this is too great an overhead both in terms of store and in terms of set up time for very short string searches. Instead a table of length 256 bits (32 bytes) is set up for each qualified string when it is decoded. This table indicates the presence or absence of each character in the string, and is used to implement the first feature of the Boyer and Moore algorithm. A useful byproduct is that some of the cost of searching for an alphabetic string in either upper or lower case can be avoided.

The code of ZED includes a store manager which works on a first fit basis, with amalgamation of contiguous free blocks. The range of sizes of block is relatively

restricted, and most blocks are small. Fragmentation problems can occur however during operations which involve opening and closing files with large blocksizes. Such operations are fortunately rare.

A stack is used to implement the procedure calling conventions, thus naturally providing facilities for recursion which are exploited when processing nested command groups.

ZED decodes and translates each line of commands into an internal data structure before starting to obey the line. This is in contrast to EDIT, which obeys commands interpretively from the textual form. Syntax errors in ZED are detected before any commands on the line have been obeyed, thus limiting the effects of typing errors more strictly.

Over 80 different error messages may be issued by ZED. These range from simple warnings about fixed field overflows to serious errors such as insufficient store. The interactive user is always given the opportunity to enter more commands following an error except after internal consistency failures (which should never occur).

CONCLUSION

This paper has described the design and implementation process which took place as a result of the decision to produce a new text editor, and has also covered the major features of the resulting program. The iterative nature of the design, coupled with prototype testing in the user community, has led to an editor which has rapidly gained wide acceptance. Indeed, the news of the existence of the final prototype versions spread so rapidly by informal means that comments were received from several dozen users, many of whom were not personally known to the author.

The main feature which distinguishes ZED is its attention to the needs of the non-interactive user. Facilities to enhance programmability have been provided in such a way that the interactive user is not unduly penalized.

ACKNOWLEDGEMENTS

Many staff and users of the University Computing Service contributed ideas and suggestions for the improvement of EDIT, which eventually spurred me into undertaking the implementation of ZED. Many more took part in the exercise of designing the new specification and trying out the prototypes. My thanks are due to all those who gave their time and effort, in particular to Arthur Norman who invented qualified strings and persuaded me that they were a good thing, and to Brian Kelk who, as well as making many suggestions about the specification, found more bugs in the prototypes than anyone else. The comments of an anonymous referee have enabled me to make a number of improvements in the text of this paper.

APPENDIX—ZED COMMANDS

Argument abbreviations:

a, b	line numbers (or . or *)
cg	command group
m, n	numbers
q	qualifier list (possibly empty)

se	search expression
s, t	strings of arbitrary characters
sw	switch value (+ or −)
name	sequence of letters (4 significant)

Qualifiers:

B	beginning
E	ending
C	control character
L	last
P	precisely
W	word
U	upper case
S	significant
N	not
n	count
[n,m]	window

Line selection

Ma	move to line a
M*	move to end-of-file
M−	move back as far as possible
M+	move forward to high water
N	move to next line
P	move to previous line
F se	find (forward)
BF se	backward find

Line insertion and deletion

Insert material may be any one of

⟨in-line text, any number of lines
terminated by Z on its own line⟩

BUFF n	to insert an in-store buffer
COPY n	to insert a copy of an in-store buffer
/s/	to insert the file with ddname or dsname s

Ia	insert before line a
Ra b	replace lines a to b
Da b	delete lines a to b
DF se	delete find
DREST	delete rest of source
DBUFFn	delete contents of buffer n
DBUFF	delete contents of all buffers

Line windowing

RFn m	restrict find
RVn m	restrict view
⟩	move character pointer to right

<	move character pointer to left
PR	reset pointer to start
PA q/s/	point after
PB q/s/	point before
EWR	end window right
EWL	end window left
EWA q/s/	end window after
EWB q/s/	end window before

Single character operations

$	force lower case
^	force upper case
← or _	change character to space
#	delete character

String operations

A q/s/t/	after
AP q/s/t/	after and point
B q/s/t/	before
BP q/s/t/	before and point
E q/s/t/	exchange
EP q/s/t/	exchange and point
DFA q/s/	delete from after
DFB q/s/	delete from before
DTA q/s/	delete till after
DTB q/s/	delete till before
LC q/s/	force lower case
UC q/s/	force upper case
'	repeat last string operation
UNDO	undo current line
SHC q/s/	show column
CC /s/	set control character

Splitting and joining

SA q/s/	split after
SB q/s/	split before
CL /s/	concatenate

Text inspection

Tn	type n lines
TLn	ditto, with line numbers
T+	type to high water
TL+	ditto, with line numbers
TBUFFn	type buffer n

File and buffer control

C /s/	commands from s
C BUFF n	commands from buffer n (destructive)

C COPY n	commands from buffer n
FROM BUFF n	select buffer n as source
TO BUFF n	select buffer n for output
TBUFFn	type contents of buffer n
DBUFFn	delete contents of buffer n
DBUFF	delete contents of all buffers
SHBUFF	show non-empty buffer numbers
TO /s/	select dsname or ddname
TO	select TO
FROM /s/	select dsname or ddname
FROM	select FROM
CF	close all (closeable) files
CF /s/	close file with ddname s

Conditionals and loops

Wherever ELSE may appear, the forms

ELIF se THEN cg ELSE cg
ELUL se THEN cg ELSE cg

may also appear.

IF se THEN cg ELSE cg
UL se THEN cg ELSE cg

IFEOF cg ELSE cg	if end-of-file
ULEOF cg ELSE cg	unless end-of-file
WH se cg	while
UT se cg	until
UTEOF cg	until end-of-file
RPT cg	repeat indefinitely
AGP	abandon group

Margins

MAn m	source margins
MAVn m	verification margins

Fixed format fields

MFn1 n2 ...	mark fields
Kn1 n2 ...	temporarily suspend fields

Jusification

JLn	set justification length
J sw	enable/disable justification
RJn m	restrict justification
VJ sw	justification verification

Global operations

GE q/s/t/	global exchange

GA q/s/t/	global after
GB q/s/t/	global before
ON se cg ELSE cg	complicated global
CGn	cancel global n
DGn	disable global n
EGn	enable global n
SHGn	show global n
VG sw	global change verification

State display

SHD	show data
SHF	show files
SHG	show globals
SHS	show switches

Termination

W	windup
Q	quit (exit command level)
STOP	sic (return code 8)

Line verification

V sw	automatic verification
VJ sw	justification verification
VG sw	global change verification
VN sw	verify line numbers
VT sw	verify line texts
VWR sw	verify right window
VWL sw	verify left window
VW sw	verify both windows
VE sw	verify edit lines
!	verify with character indications
?	verify current line

Procedures

PROC name cg	define
SHPROC name	show
CPROC name	cancel
DO name	obey

Hexadecimal mode

X sw	switch hexadecimal mode on/off

Miscellaneous

Z /s/	set terminator string
TR sw	trailing spaces
SQ sw	sequence numbers in output
SQS sw	source sequence numbers
Ha	halt at line a

COMM /s/	comment to verification file
COLS COLS n COLS +	} verify column numbers
CS sw	compress spaces
= n	set line number
BRK sw	enable/disable BREAK
REWIND	sic
MXLLn	max line length
SCC sw	suppress control characters
SO sw	suppress overflow errors
FLUSH	flush output queue
STACK n	set stack size to n bytes
MXSS n	set maximum store size to n bytes
STORE	show store usage
MS sw	monitor store usage

REFERENCES

1. P. Hazel, 'A general purpose text editor for OS/360', *Software—Practice and Experience*, **4**, 389–399 (1974).
2. S. R. Bourne, 'A design for a text editor', *Software—Practice and Experience*, **1**, 73–82 (1971).
3. *Cambridge 370/165 Users' Reference Manual*, 4th edn. University of Cambridge Computing Service (1979).
4. R. S. Boyer and J. S. Moore. 'A fast string searching algorithm', *Comm. ACM*, **20**, 762–772 (1977).
5. P. Hazel, 'The development of text editing at Cambridge', *IUCC Bulletin*, **1**, No. 3, Inter-University Committee on Computing (1979).